University of Waterloo
Department of Electrical and Computer Engineering
ECE 250 – Algorithms and Data Structures

Midterm Examination (11 pages)
Instructor: Douglas Harder
February 17, 2004 17:30-19:00

Name (last, first)	Student ID

Do problems 1-7. The last question is a bonus.

The number in brackets denotes the weight of the question. If information appears to be missing from a problem, make a reasonable assumption, state it, and proceed. If the space to answer a question is insufficient, use the back of the previous page. You may use diagrams to supplement (but not replace) sentence answers.

Closed book. No calculators.

Question	Mark
1	/10
2	/12
3	/10
4	/6
5	/12
6	/10
7	/6
Bonus	/3
Total	/66

Algorithm Analysis

1. **[10]** Use the detailed model, that is, use τ_+, τ_{fetch}, etc., to determine the run-time of the body of the following method. Use the given table and do not total the run times.
You may use τ_F, τ_S, τ_C, τ_R, and τ_N, in place of τ_{fetch}, τ_{store}, τ_{call}, τ_{return}, and τ_{new}, respectively.

You may wish to recall that the time it takes to fetch the instance variable `array.length` is equal to the time it takes to fetch a local variable.

```
public void average( int[] array ) {
1     int sum = 0;

2     if ( array.length == 0 )
3          return 0;

4     for ( int i = 0; i < array.length; i ++ )
5          sum += array[i];

6     return sum / array.length;
}
```

Line	Case 1 (describe)	Case 2 (describe)
1		
2		
3		
4a		
4b		
4c		
5		
6		

Asymptotic Analysis

2a. **[3]** From the definition of big-O, show that $4n^2 + n - 1 = O(2n^2)$.

2b. **[2]** Show that $\ln(n) = \Theta(\log_2(n))$, that is, $\ln(n)$ and $\log_2(n)$ are big-O of each other.

2c. **[2]** Suppose $\lim_{n=\infty} \frac{f(n)}{g(n)} = \infty$. For each of these statements, circle *true* if is correct, and *false* otherwise.

$f(n) = O(g(n))$	true	false
$g(n) = O(f(n))$	true	false

2d. **[3]** Prove that $x^n = O(e^x)$ where n is a fixed positive integer.

2e. **[2]** Simplify the expression

$$O(1) + O(n)\,T(f) + O(n \ln(n)) + O(n)$$

where $T(f)$ is the time it takes to make a call to the method f.

Foundational Data Structures

3. **[5+5]** Write the code for the methods `prepend`, which inserts an object into the start of the linked list; and `getCount`, which counts the number of elements in the linked list and returns that value.

```
public class LinkedList {
      protected Element head;
      protected Element tail;

      public final class Element {
            Object datum;
            Element next;

            public Element( Object datum, Element next ) {
                  this.datum = datum;
                  this.next = next;
            }
            // ...
      }
      // ...
      public void prepend( Object obj ) {














      }
      public int getCount() {














      }
}
```

Abstract Data Types

4. **[6]** A container is known to have zero or more objects of the wrapper class `Double`. Write a method `average` which takes such a container as an argument, finds the average of all the elements, and returns that average. Your method should call the `getEnumeration` method of the container and use the returned enumeration to access the elements in the container. If the container is empty, your method should throw the `ContainerEmptyException` exception. You need not check that the elements in the container are instances of the class `Double`.

```
public interface Enumeration {
      public boolean hasMoreElements();
      public Object nextElement();
}
public class Double {
      double value;
      // ...
      public double doubleValue() {
            return value;
      }
      // ...
}
public SomeClass { // ...
  public double average( Container c ) {











  }
}
```

Stacks and Queues

5a. **[3+3]** Suppose the class `StackAsLinkedList` implements a stack using a singly-linked list which contains a head reference and a tail reference. For example:

```
public class LinkedList {
     protected Element head, tail;

     public class Element {
          Object datum;
          Element next;
          // ...
     }
     // ...
}
```

The class `LinkedList` has two methods which may be used to insert an element into the linked list: `append`, which inserts a new element at the tail of the linked list; and `prepend`, which inserts a new element at the head of the linked list. Therefore, there are two possible implementations of the `push` method of a stack:

```
public void pushAppend( Object o ) {
     list.append( o );
}
public void pushPrepend( Object o ) {
     list.prepend( o );
}
```

If the method `pushAppend` is chosen to be used in the implementation of the `push` method of the class `StackAsLinkedList`, what must the run time, using asymptotic notation, of the `pop` method for a stack containing *n* elements? Why (give one sentence)?

If the method `pushPrepend` is chosen, what is the run time, using asymptotic notation, of the `pop` method for a stack containing *n* elements? Why (give one sentence)?

5b. **[6]** For the class `AbstractQueue`, write the method `appendQueue` which takes as its argument any object which implements the `Queue` interface. This method should dequeue the elements in the argument queue and enqueue them in the current queue until either the argument queue is empty or `this` queue is full. When this method returns, all elements in the argument queue must either be in `this` queue (in the same order in which they appeared in the argument queue) or must be in argument queue (again in the order in which they were placed into the queue).

```
public abstract class AbstractQueue
            extends AbstractContainer implements Queue {
  // ...
  public void appendQueue( Queue q ) {














  }
}
public interface Queue extends Container {
      Object getHead();
      void enqueue( Object object );
      Object dequeue();
}
public interface Container extends Comparable {
      int getCount();
      boolean isEmpty();
      boolean isFull();
      void purge();
      void accept ( Visitor visitor );
      Enumeration getEnumeration();
}
```

Ordered and Sorted Lists

6a. **[3]** Circle which elements would be examined in the following implementation of a sorted list of integers when determining if the value 42 is stored in this array.

1	3	9	12	23	25	32	33	35	39	42	52	59	61	73

6b. **[2]** Why can you not perform a binary search on a sorted list which is implemented as a linked list? Use one or two sentences.

6c. **[3]** A binary search runs in O(ln(n)) time where n is the number of elements in the sorted list. You may find this by solving the recurrence relation

$T(n) = T((n - 1)/2) + 1$, and $T(1) = 1$.

Suppose instead you have an algorithm which, at each step, checks the midpoint of not one, but both halves of the sorted array. The recurrence relation for such a problem is given by

$T(n) = 2\, T((n - 1)/2) + 1$, and $T(1) = 1$.

Assuming that $n = 2^k - 1$, what is the run time of such an algorithm? Your answer should be sufficiently simple to justify your next answer.

6d. **[2]** Justify conceptually why the solution to the previous recurrence relation makes sense. Use one or two sentences.

Project 2

Consider the following implementation of method `SetCoefficient` in the class `PolynomialAsArray`. The array of polynomial coefficients is stored in the instance variable `coeffs`.

```
public void SetCoefficient( int i, double c ) {
1     if ( i < 0 ) throw new RuntimeException();
2     elif ( i < this.getDegree() ) coeffs[i] = c;
3a    elif ( i == this.getDegree() ) {
3b         coeffs[i] = c;
3c         if ( c == 0 )
3d              resize_coeffs();  // shrink the array
4a    } else if ( c != 0 ) {  // i > this.getDegree()
4b         double[] tmp = new double[i + 1];
4c         tmp[i] = c;
4d         for ( int j = 0; j <= this.getDegree(); j++ )
4e              tmp[j] = this.coeffs[j];
4f         coeffs = tmp;
      }
}
```

What is the runtime of the following two implementations of the `assign` method which sets the current polynomial equal to the argument polynomial `p`? You should refer to the code above, but you need not do more than a cursory asymptotic analysis.
You may use *n* to represent the degree of the polynomial `p`.

```
public void assign( Polynomial p ) {
     coeffs = new double[0];

     for ( int i = 0; i <= p.getDegree(); i++ )
          this.setCoefficient( i, p.getCoefficient( i ) );
}
```

7a. **[3]**

```
public void assign( Polynomial p ) {
      coeffs = new double[0];

      for ( int i = p.getDegree(); i >= 0; i-- )
            this.setCoefficient( i, p.getCoefficient( i ) );
}
```
7b. **[3]**

Bonus. **[3]** If *n* is the maximum of the degree of `this` polynomial and the degree of the polynomial `p`, then the average runtime for the following method is O(*n*). Under what conditions on `this` and `p` will the runtime of this algorithm be O(n^2)?

```
public Polynomial add( polynomial p ) {
      Polynomial q = new PolynomialAsArray();
      Polynomial min, max;

      if ( p.getDegree() > this.getDegree() ) {
            max = p;
            min = this;
      } else {
            max = this;
            min = p;
      }

      for ( int i = max.getDegree(); i >= 0; i-- ) {
            q.setCoefficient( i, max.getCoefficient(i) );
      }

      for ( int i = min.getDegree(); i >= 0; i-- ) {
            q.setCoefficient( i,
                  min.getCoefficient(i) + q.getCoefficient(i)
            );
      }

      return q;
}
```

Additional Information

$$\frac{d}{dx}e^x = e^x \quad \frac{d}{dx}\ln(x) = \frac{1}{x} \quad \frac{d}{dx}x^n = nx^{n-1} \quad \frac{d^n}{dx^n}x^n = n! \quad \frac{(2^m - 1) - 1}{2} = 2^{n-1} - 1$$

τ_{fetch}, τ_{store}, τ_{+}, τ_{-}, τ_{*}, $\tau_{/}$, $\tau_{<}$, τ_{call}, τ_{return}, $\tau_{[\,]}$, τ_{new}

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{n} i^3 = \frac{n^2(n^2+1)}{4}$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$\sum_{i=0}^{n} ir^i = \frac{(rn - n - 1)r^{n+1} + r}{(r-1)^2}$$